

Extended Authorization Framework

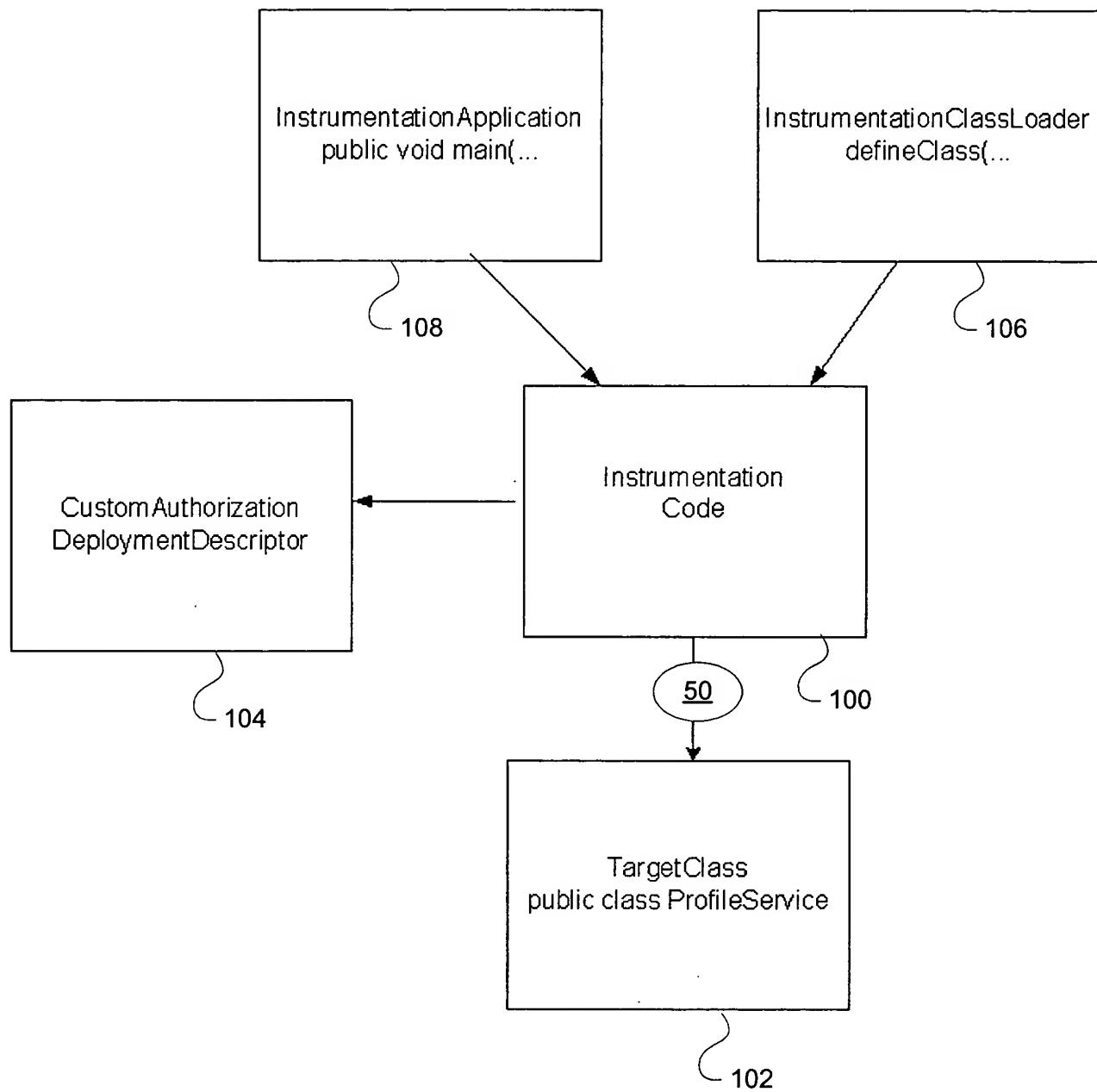


Fig. 1

Target Class ProfileService

```
public Profile getProfile(String profileName, String attributes)
{
    System.out.println("\n*** ProfileService.getProfile entry ***");
    //get attributes associated w/profileName from DB
}
```

102a

Fig. 2

Custom Authorization Deployment Descriptor

```
...
<authorizationTarget class="ProfileService">
    <subjectFactory class="ThreadSubjectFactory"/>
    <method name="getProfile">
        <requiredPermission class ="com.ibm.resource.security.auth.ProfilePermission">
            <property name="name" value="UserProfile" />
            <property name="actions" value="Read" />
            <property name="attributes" value="attributes" scope="local" />
        </requiredPermission>
        <privilegedAction class ="ReadAction">
            <property name="attributes" value="attributes" scope="local" />
        </privilegedAction>
    </method>
</authorizationTarget>
...
```

104a

Fig. 3

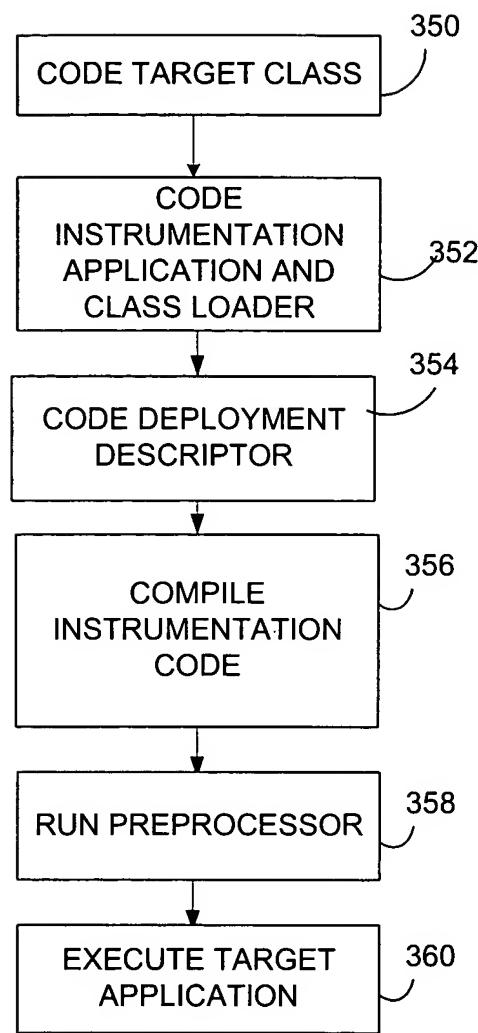


Fig. 4

Target Class ProfileService — Instrumented Code

```
public Profile getProfile(String profileName, String attributes)
{
    System.out.println("\n*** ProfileService.getProfile entry ***");

    /* JAVA LANGUAGE EQUIVALENT OF INSTRUMENTATION
     (It's actually inserted into the target as byte code)
    */
    SubjectFactory subjectFactory= new ThreadSubjectFactory();
    Subject user =subjectFactory.getSubject();

    if(user != null && (System.getProperty("custom_authorization") != null))
    {

        PermissionFactory permissionFactory = (new DefaultPermissionFactory()).getFactory();

        permissionFactory.setProperty("RequiredPermission", "com.ibm.resource.security.auth.ProfilePermission");
        permissionFactory.setProperty("RequiredPermission.name", "UserProfile");
        permissionFactory.setProperty("RequiredPermission.actions", "read");
        permissionFactory.setProperty("RequiredPermission.attributes", attributes);

        PrivilegedActionFactory privilegedActionFactory = (new DefaultPrivilegedActionFactory()).getFactory();
        privilegedActionFactory.setProperty("PermissionFactory", permissionFactory);
        privilegedActionFactory.setProperty("PrivilegedAction", "ReadAction");
        privilegedActionFactory.setProperty("attributes", attributes);

        Subject.doAsPrivileged(user, privilegedActionFactory.getPrivilegedAction(), null);
    }
    /*END INSTRUMENTATION */

    //get attributes associated w/profileName from DB
}
```

102b

Fig. 5

Byte-code instrumentation could be achieved using BCEL

```

SubjectFactory subjectFactory= new ThreadSubjectFactory();
// 6 16:new      #6 <Class ThreadSubjectFactory>
                                6)CONSTANT_Class[7](name_index = 71)
                                ...
                                71)CONSTANT_Utf8[1]("ThreadSubjectFact
                                ory")
                                bcel:
                                subjectFactoryClass_idx =
                                constPool.addClass("ThreadSubjectFactory");
                                InstructionList patchEnd = new InstructionList();
                                patchEnd.append(new NEW(subjectFactoryClass_idx));

// 7 19:dup
                                bcel:
                                patchEnd.append(new DUP0);

// 8 20:invokepecial #7 <Method void ThreadSubjectFactory()>
                                7)CONSTANT_Methodref[10](class_index =
                                6, name_and_type_index = 64)
                                ...
                                6)CONSTANT_Class[7](name_index = 71)
                                ...
                                71)CONSTANT_Utf8[1]("ThreadSubjectFact
                                ory")
                                ...
                                64)CONSTANT_NameAndType[12](name_in
                                dex = 42, signature_index = 43)
                                ...
                                42)CONSTANT_Utf8[1]("<init>")
                                43)CONSTANT_Utf8[1]("(O)V")
                                (invoke special must pop the object
                                reference of the stack...)
                                bcel:
                                subjectFactoryConstructor_idx =
                                constPool.addMethodref("ThreadSubjectFactory", "<init>",
                                "(O)V");
                                patchEnd.append(new NEW
                                INVOKESPECIAL(subjectFactoryConstructor_idx));

```

The diagram illustrates the flow of byte code instructions starting from address 600. Address 600 is at the top left, with three arrows pointing downwards to three separate code snippets. The first snippet is for the new instruction, the second for dup, and the third for invokepecial.

- new:** This path starts with the instruction // 6 16:new. It then shows the generation of a CONSTANT_Class instruction (6) and a CONSTANT_Utf8 instruction (71). Below this, BCEL code is shown for adding the class to the constant pool and creating a NEW instruction.
- dup:** This path starts with the instruction // 7 19:dup. It shows the generation of a CONSTANT_Methodref instruction (7) and a CONSTANT_Class instruction (6). Below this, BCEL code is shown for adding the method to the constant pool and creating a DUP0 instruction.
- invokepecial:** This path starts with the instruction // 8 20:invokepecial #7 <Method void ThreadSubjectFactory()>. It shows the generation of a CONSTANT_NameAndType instruction (64), followed by CONSTANT_Utf8 instructions for the method name and signature. Below this, BCEL code is shown for adding the method to the constant pool and creating an INVOKESPECIAL instruction.

102c

Fig. 6